Evolution vs. Intelligent Design in Consensus Protocols

Yee Jiun Song, Robbert van Renesse, Fred B. Schneider
Cornell University

Danny Dolev
Hebrew University of Jerusalem

***Abstract*— Consensus is an important building block for building replicated systems, and many consensus protocols have been proposed. In this paper, we show that many consensus protocols can be derived from the same simple *genes*. We present these genes in the form of a skeleton algorithm that can be configured to produce, among others, three well-known consensus protocols: Paxos, Chandra-Toueg, and Ben-Or. Although each of these protocols specify only one quorum system explicity, we show that all employ a second quorum system. We use the skeleton algorithm to implement a replicated service, allowing us to compare the performance of these consensus protocols under various workloads and failure scenarios. From this we learn, for example, that weak leader election in Paxos unnecessarily causes performance degradation in certain failure scenarios.**

I. Introduction

Computers will fail, and for many systems it is imperative that such failures be tolerated. Replication, the general approach toward fault tolerance, requires a protocol for replicas to agree on values or actions. The original *agreement* or *consensus* problem was proposed in [1]. Many versions of the problem and corresponding solutions have been introduced since (see [2] for a survey of just the first decade, containing well over 100 references).

This paper focuses on protocols for Internet-like systems in which there are no real-time bounds on execution or message latency. Such systems are often termed *asynchronous*. While published asynchronous consensus protocols may at first appear complex and quite different from each other, we claim that all these protocols are derived from the same simple *genes*.

This paper makes three contributions:

- We present the genes of consensus algorithms in the form of a skeleton algorithm that can be configured to produce various consensus protocols. The skeleton algorithm gives insight into how consensus protocols work, and we learn that consensus requires not one but two separate quorum systems;
- We demonstrate how this approach can be used to instantiate three well-known consensus protocols: Paxos [3], Chandra-Toueg [4], and Ben-Or [5];
- We implement our approach and present a performance comparison of these protocols under varying workload and crash failures. We learn interesting trade-offs between various design choices in consensus algorithms.

The rest of this paper is organized as follows. Section II describes the consensus problem and proposes terminology for discussing various consensus protocols. Before we present our skeleton, we need to introduce *extended quorum systems* as a building block in Section III. Section IV describes the skeleton, while Section V illustrates instantiations. Section VI describes the implementation of the skeleton and compares three well-known consensus protocols. Section VII concludes. Appendices I and II contain correctness proofs.

II. The Consensus Problem

In prior work, there has been neither agreement nor consensus on terminology. We propose nomenclature for talking about the consensus problem and protocols to solve it (see Table I).

In the consensus problem there is a set of *proposers*, each of which can propose a *proposal*, and a set of *deciders*, each of which *decides* one of the proposals. The goal is to ensure each non-faulty decider decides the same proposal, even in the face of faulty proposers.

We must specify the execution and failure model of *nodes* (computers that run programs) and *links* (network connections between nodes). Nodes run *actors*, which are either proposers or deciders. A node may run both a proposer and a decider—in practice, the proposer often would like to learn the outcome of the agreement.

Nodes are either *honest*, executing programs faithfully, or *Byzantine* [6], exhibiting arbitrary behavior. We will also use the terms *correct* and *faulty*, but not as alternatives to honest and Byzantine. A correct node is an honest node that always eventually makes progress. A faulty node is a Byzantine node or an honest node that has crashed or will eventually crash. Note that honest and Byzantine are mutually exclusive, as are correct and faulty. However, a node can be both honest and faulty.

We assume that each pair of nodes is connected by a *link*, which is a bi-directional reliable virtual circuit that ensures messages sent on this link are delivered, eventually, and in the order in which they were sent (*i.e.*, an honest sender keeps retransmitting a message until it receives an acknowledgment or crashes). Also, the receiver can tell who sent a message (*e.g.*, using MACs), so a Byzantine node cannot forge a message so it appears like a message sent by an honest node.

Because our system is asynchronous, we do not assume timing bounds on execution of programs or on latency of communication. We also do not assume that a node on one side of a link can determine whether the node on the other side of the link is correct or faulty. Timeouts cannot reliably detect faulty nodes in an asynchronous system, even if only crash failures are allowed.

Why is consensus hard? Consider the following strawman protocol: each decider collects proposals from all proposers, determines the minimum proposal from among the proposals it receives (in case it received multiple proposals), and decides on that one. If there were no faulty nodes at all, such a protocol would work, albeit limited in speed by the slowest node or link.

Unfortunately, even if only crash failures are possible, deciders do not know how long to wait for proposers. If deciders were to use time-outs, they might time-out on different proposers, and these deciders would decide different proposals as a result. Thus each decider has no choice but to wait until it has received a proposal from all proposers. If one of the proposers is faulty, such a decider will never decide.

In an asynchronous system with crash failures (Byzantine failures comprise crash failures), there exists no deterministic protocol in which all correct deciders eventually decide [7] (a result called FLP after the people that showed this impossibility, Fisher, Lynch, and Patterson). We can circumvent this limitation by not requiring that all correct deciders eventually decide. Instead, we will require only that the consensus protocol cannot reach a state in which some correct decider can never decide. The strawman protocol of deciding the minimum proposal can reach a state in which deciders wait indefinitely for a faulty proposer, and is, therefore, not a consensus protocol, even with our weaker requirement.

A protocol that solves the consensus problem must have the following three properties:

Definition: Agreement: If two honest deciders decide, they decide the same proposal.

Definition: Validity: If all honest proposers propose the same proposal v, then an honest decider that decides will decide v.

Definition: Non-Blocking: Given any run of the protocol that reaches a state in which a particular correct decider has not yet decided, there exists a continuation of the run in which that decider does decide on a proposal.

To abstract the interaction that takes place in agreement algorithms we describe a mechanism through

term	*description aka*
actor	proposer, decider, selector, or registrar
Byzantine	potentially exhibits arbitrary behavior
correct	honest and makes progress eventually
crashed	halted execution indefinitely
decider	actor who wants to decide
faulty	Byzantine or stops making progress
guarded set	contains at least one honest participant
honest	not Byzantine (but may crash)
instance	phase in protocol
(network) link	connects two participants
node	hardware on which actor runs
participant	member of a quorum system
proposal	initial value submitted to protocol
proposer	actor whose role is to propose a value
quorum (set)	any two quorums of participants intersect
quorum system	structure on set of actors
registrar	actor that prevents multiple decisions
selector	actor that suggests decisions
suggestion	proposal + instance identifier
max-wait set	maximal set of participants one can wait for

TABLE I
TERMINOLOGY.

which messages are exchanged. The mechanism is based on the notion of *quorum systems*, which we now review and extend.

III. Extended Quorum Systems

Before we introduce our skeleton algorithm, we introduce a useful building block. An *extended quorum system* is a quadruple $(\mathcal{P}, \mathcal{M}, \mathcal{Q}, \mathcal{G})$. $\mathcal{P}$ is a set of nodes called the *participants*. $\mathcal{M}, \mathcal{Q}$, and $\mathcal{G}$ are each a collection of subsets of participants (that is, each is a subset of $2^{\mathcal{P}}$). $\mathcal{M}$ is the collection of *maximal-wait sets*, $\mathcal{Q}$ the collection of *quorum sets*, and $\mathcal{G}$ the collection of *guarded sets*. Each is defined below.

A subset of $\mathcal{P}$ is a *guarded set* if and only if it is guaranteed to contain at least one honest participant. Note that a guarded set may consist of a single participant that may be crashed but is not Byzantine. As an example, consider an (n, t)*-threshold quorum system*, a system with n participants, of which at most t are faulty. In the case of crash failures only, the guarded sets are all non-empty subsets of $\mathcal{P}$. In the Byzantine case, guarded sets have to be of size larger than t in order to guarantee that they contain at least one honest participant.

When requesting information from all participants, crashed or Byzantine participants may never respond. An actor often tries to collect as many responses to a broadcast request as possible, but has to stop collecting additional responses when it is in danger of waiting indefinitely. $\mathcal{M}$ characterizes this—it is a set of subsets of $\mathcal{P}$, none contained in another, such that some $M \in \mathcal{M}$ contains all the correct nodes.[1] In a threshold quorum system, the maximal-wait sets are all subsets of $n - t$ participants, where n is the number of nodes and t is the maximum number of failures.

Quorums are sets of nodes that satisfy Consistency:

Definition: Consistency: An extended quorum system satisfies *Consistency* if and only if the intersection of any two quorums (including a quorum with itself) is guaranteed to contain an honest participant. (In other words, the intersection of two quorums is a guarded set.)

To illustrate, consider how consistency can be satisfied in an (n, t)-threshold system. With only crash failures possible, consistency requires that quorums be larger than $n/2$ participants, because two strict majorities always intersect, and no participants are Byzantine. When Byzantine failures are possible, consistency requires having quorums be larger than size $(n + t)/2$. To see why this works, consider any two quorums. Without removing duplicates, the sets together have more than $n + t$ elements. But there are only n participants, so the intersection must contain more than t participants, and thus contains at least one honest participant.

Definition: Availability: An extended quorum system satisfies *Availability* iff every maximal-wait set contains a quorum.

Because Availability requires that every maximal-wait set contains a quorum, we get that $n - t > n/2$, or $n > 2t$, putting a lower bound on n. We obtain that $n - t > (n + t)/2$, or $n > 3t$.

[1] For those familiar with Byzantine Quorum Systems [8], $\mathcal{M}$ is the set of complements of the fail-prone system $\mathcal{B}$. For the purposes of this paper, it is often more convenient to talk about maximal-wait sets.

	Crash	Byzantine
guarded set (in $\mathcal{G}$)	> 0	$> t$
quorum set (in $\mathcal{Q}$)	$> n/2$	$> (n+t)/2$
maximal-wait set (in $\mathcal{M}$)	$= n - t$	$= n - t$
set of participants ($\mathcal{P}$)	$> 2t$	$> 5t$

TABLE II
SIZE REQUIREMENTS FOR THRESHOLD QUORUM SYSTEMS THAT SATISFY CONSISTENCY AND OPAQUENESS.

Availability requires that quorums can have no more than $n - t$ elements, otherwise there would exist maximal-wait sets that do not contain a quorum.

In this paper, we will also require that extended quorum systems satisfy *Opaqueness* [8]:[2]

Definition: Opaqueness: An extended quorum system satisfies *Opaqueness* if and only if each maximal-wait set contains a quorum consisting entirely of honest participants.

We illustrate what this requirement means for (n, t)-threshold quorum systems. In a crash failure model, opaqueness requires that every maximal-wait set contains a quorum, so we get that $n-t > n/2$, or $n > 2t$, placing a lower bound on n. In the Byzantine model, t of the participants in a maximal-wait set may be Byzantine. Thus we have to require that $n - 2t > (n+t)/2$, which simplifies to $n > 5t$. Table II summarizes requirements for $\mathcal{P}$, $\mathcal{M}$, $\mathcal{Q}$, and $\mathcal{G}$ in (n, t)-threshold systems.

The simplest example of quorum systems are threshold quorum systems, but other quorum systems may be more appropriate for particular applications. See [9] and [8] for advantages and disadvantages of various quorum systems for crash and arbitrary failure models respectively.

One degenerate extended quorum system, used in some well-known consensus protocols, is a *leader extended quorum system*: There is one participant (the leader), and that participant by itself forms the only maximal-wait set in $\mathcal{M}$, quorum in $\mathcal{Q}$, and guarded set in $\mathcal{G}$. Because quorum sets have to satisfy consistency, the leader has to be honest.

[2]Opaqueness is a stronger property than *Availability* typically required of quorum systems, as Availability only requires that each maximal-wait set contain a quorum, possibly including Byzantine members.

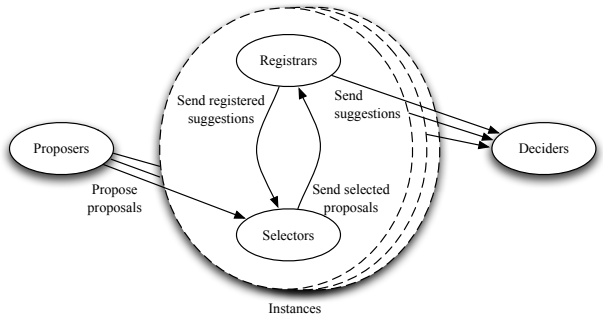

Fig. 1. The actions of the various actors.

IV. THE CONSENSUS SKELETON

In Section II, we presented a strawman consensus protocol that, in the presence of faulty nodes, may reach a state where deciders can never decide. To avoid this, consensus protocols invoke multiple *instances*, where an instance is an execution of a protocol that, once started, runs in isolation. Instances have also been called *rounds*, *phases*, or *ballots*.

In order to guarantee consistency among decisions of deciders in the presence of multiple instances, we introduce two new types of actors in addition to proposers and deciders, namely *registrars* and *selectors*.[3] A proposer sends its proposal to the selectors. Selectors and registrars exchange messages and occasionally registrars inform deciders about potential values for decision. Deciders apply some filter to reach a decision.

The actions of the various actors are summarized in Figure 1. Broadly speaking, the objective of selectors is to reach a decision within an instance, while the objective of registrars is to maintain a collective memory that ensures that decisions are maintained across instances, preventing conflicting decisions.

A. Instances

An instance decides a proposal if an honest decider decides a proposal in that instance. All honest deciders

[3]As stated before, a node may run multiple actors, although each can run at most one registrar and at most one selector.

that decide in an instance must be guaranteed to decide the same proposal, so an instance cannot decide multiple proposals. It is not guaranteed that an instance decides any proposal. By having multiple instances, if one instance does not decide, then future ones may decide. It is important to guarantee that if multiple instances decide, they decide the same proposal.

Instances are identified by instance identifiers r from a totally ordered set that we will call $\bar{\mathbb{N}}$ (can be, but does not have to be, $\mathbb{N}$). Instance identifiers induce an ordering on instances, and we say that one instance is *before* or *after* another instance, but keep in mind that instances may execute concurrently.

We name proposals $v, w, ...$ Within an instance, proposals are paired with instance identifiers. We call a pair (r, v) a *suggestion*, where v is the proposal and r an instance identifier. A special suggestion $\perp$ is used to indicate absence of a specific proposal.

Selectors *select* proposals, and registrars *register* suggestions. Each registrar keeps careful track of the last suggestion that it has registered. The initial registered suggestion of a registrar is $\perp$.

A new instance starts with the registrars sending their current registered suggestions to the selectors. (Exactly how an instance starts depends on the complete design of the consensus protocol, and will be addressed later.) Each selector determines if one of the suggestions it receives could have been decided in a previous instance. If so, it selects the corresponding proposal (of which there can be at most one). If not, it selects one of the proposals issued by the proposers. The selector creates a suggestion from the selected proposal using the current instance identifier, and sends the suggestion to the registrars.

If a registrar receives the same suggestion from a quorum of selectors (that is, all selectors in the quorum sent the same instance identifier and proposal), it (i) registers the suggestion, and (ii) broadcasts the suggestion to the deciders. If a decider receives the same suggestion from a quorum of registrars, the decider decides the corresponding proposal in those suggestions.

B. Guarded Proposal

Selectors have to be careful not to select proposals that could conflict with prior decisions. Before selecting a proposal in an instance, a selector obtains a set of suggestions L from each participant in a maximal-wait set of registrars. A proposal v is a *potential-proposal* if L contains suggestions containing v from a guarded set. This means that at least one honest registrar sent a suggestion containing v. The selector computes the *guarded proposal* of L, if any, as follows:

1) Consider each potential-proposal v separately:
   a) Consider all subsets of suggestions containing v from guarded sets of registrars. The minimum instance identifier in a subset is called a *guarded-instance-identifier*;
   b) The maximum among the guarded-instance-identifiers for v is called the *associated-instance-identifier* of v. (Note that because v is a potential-proposal, there has to be at least one guarded-instance-identifier and thus the maximum is well-defined.) The *support-sets* for v are those subsets of suggestions for which the guarded-instance-identifier is the associated-instance-identifier;
2) Among the potential-proposals, select all proposals with the maximal associated-instance-identifier. If there is exactly one such potential-proposal v', and $v' \neq \perp$, then this is the guarded proposal. Otherwise there is no guarded proposal.

For example, consider a Byzantine threshold quorum system $\mathcal{Q}$ with $t = 2$ and $n = 11$. Thus, a quorum has to have more than $(n + t)/2$ elements, which is 7 or more in our case, and a maximal-wait set has to contain at least $n - t = 9$ elements. A guarded set has at least $t + 1 = 3$ elements. Now consider the following suggestions in the maximal-wait set: 4 suggestions with the value $(3, \texttt{green})$, 2 suggestions with the value $(5, \texttt{green})$, and 3 suggestions with the value $(4, \texttt{red})$. Both `green` and `red` are potential-proposals. The maximum instance identifier among sets of 3 suggestions for `green` is 3, as there are only 2 suggestions with instance identifier 5. The maximum

instance identifier among sets of 3 suggestions for `red` (there is only one such set) is 4, thus 4 is the maximum associated-instance-identifier. Because `red` is the only proposal with an associated-instance-identifier of 4, `red` is the guarded proposal even though there are twice as many suggestions supporting `green`!

For benign systems, the guarded proposal in an (n, t)-threshold quorum system is simply the proposal corresponding to the maximum instance identifier in L. For Byzantine systems, divide the suggestions in L by proposal, and sort each subset by instance identifier. Consider the proposal with the highest $t + 1^{st}$ instance identifier. If there is exactly one of these, this is the guarded proposal.

Note that in protocols in which instances are invoked one after the other completes, sequentially, *associated-instance-identifier* will always be the identifier of the previous instance.

We prove in Appendix I that if a decider obtains suggestions (r, v) from a quorum of registrars (and consequently decides), then any honest selectors in instances $\geq r$ are guaranteed to compute a guarded proposal v' such that $v' = v$ (unless they crash). If a selector fails to compute a guarded proposal in a particular instance then this is both evidence that no prior instance can have decided and a guarantee that no prior instance will ever decide. However, the reverse is not true. If a selector computes a guarded proposal v', it is not guaranteed that v' is or will be decided.

C. Extended Quorum System API

Participants of an extended quorum system send and receive messages of the form $\langle$*message-type*, *instance*, *source*, *suggestion*$\rangle$. The source indicates the sending node.

An extended quorum system $\mathcal{E} = (\mathcal{P}, \mathcal{M}, \mathcal{Q}, \mathcal{G})$ has the following interface:

- $\mathcal{E}$.*broadcast*(m): send a message m to all participants in $\mathcal{P}$;
- $\mathcal{E}$.*wait*(*pattern*): wait for messages, matching the given pattern (specifies, for example, the message type and instance number). When the sources of the collected messages form an element or a superset of an element of $\mathcal{M}$, then return the set of collected messages;
- $\mathcal{E}$.*uni-quorum*(set of messages): if the set of messages contains the same suggestion from a quorum, then return that suggestion.[4] Otherwise, return $\perp$;
- $\mathcal{E}$.*guarded-proposal*(set of messages): return the guarded proposal among these messages, or $\perp$ if there is none.

D. State Maintenance

Each selector and registrar is associated with an instance, which is the instance it is currently sending messages in, and receiving messages from. A registrar can change the instance with which it is associated, progressing to a later instance after aborting the current one. A selector can also progress to later instances, but unlike a registrar it is allowed to keep participating in older instances.

The instance protocol uses *not one but two extended quorum systems*:

1) Registrars form an extended quorum system $\mathcal{R}$ that is the same for all instances. $\mathcal{R}$ has to satisfy consistency and opaqueness. Selectors use $\mathcal{R}$ to find the guarded proposal, if any, to select proposals that do not conflict with earlier decisions.
2) Selectors form an extended quorum system $\mathcal{S}^r$, which may be different for each instance r. Each $\mathcal{S}^r$ has to satisfy both consistency and opaqueness as well. Registrars in instance r use quorums of $\mathcal{S}^r$ to avoid having two registrars register different suggestions within the same instance.

Deciders, although technically not part of an instance, do try to obtain the same suggestion from a quorum of registrars in each instance. We associate deciders with instances for simplicity of presentation. Also for convenience, we will have deciders form an extended quorum system $\mathcal{D}$. However, deciders never send messages, and thus we will only use $\mathcal{D}$ to send messages to all deciders.

[4] By the consistency property, there can be at most one such suggestion.

Each selector i maintains a set P_i containing proposals it received (across instances). A selector waits for at least one proposal before participating in the rest of the protocol, so P_i is never empty during execution of the protocol. (Typically, P_i first contains a proposal from the proposer on the same node as selector i.) For simplicity we assume an honest proposer sends a single proposal. The details of how P_i is formed and used are different for different agreement protocols, so this will be discussed when the full protocols are presented. P_i has an operation $P_i.pick(r)$ that returns either a single proposal from the set or some value as a function of r. Different protocols use different approaches for selecting that value, and these will also be discussed later.

Registrars are assumed to have state that survives crashes and recoveries. In particular, a registrar j running on an honest node maintains:

r_j: current instance identifier;
c_j: last registered suggestion, initially $\perp$.

Both increase monotonically over time.

E. The Instance Mechanism

A protocol execution is a collection of instance executions. Figure 2 shows the mechanism that enables the execution of an individual instance. We call this the *instance mechanism*. A protocol execution starts when the first instance execution starts. How instances are invoked is different for different protocols, and some ways are described in Section V. The extended quorum systems used in the instance protocol determines the potential participants in each instance and is also protocol specific. Potential participants execute in an instance once preconditions for their actions are satisfied.

Lemma 1: In the mechanism in Figure 2,

(a) if any honest registrar i computes a suggestion $q_i^r \neq \perp$ in Step (R.2) of instance r, then any honest registrar that computes a non-$\perp$ suggestion in that step of that instance, computes the same suggestion.
(b) if any honest decider k computes a suggestion $d_k^r \neq \perp$ in Step (D.2) of instance r, then any honest decider that computes a non-$\perp$ suggestion in that step of that instance, computes the same suggestion.

At the **start of instance** r, each **registrar** i executes:

(R.0) send c_i to all participants (selectors) in $\mathcal{S}^r$:
$\mathcal{S}^r.broadcast(\langle \texttt{select}, r, i, c_i \rangle)$

Each **selector** j in $\mathcal{S}^r$ executes:

(S.1) wait for `select` messages from registrars:
$L_j^r := \mathcal{R}.wait(\langle \texttt{select}, r, *, * \rangle)$;
(S.2) see if there is a guarded proposal:
$v_j^r := \mathcal{R}.guarded\text{-}proposal(L_j^r)$;
(S.3) if not, select from received proposals instead:
if $v_j^r = \perp$ **then** $v_j^r := P_j.pick(r)$ **fi**;
(S.4) send a suggestion to all registrars:
$\mathcal{R}.broadcast(\langle \texttt{register}, r, j, (r, v_j^r) \rangle)$;

Each **registrar** i (still in instance r) executes:

(R.1) wait for `register` messages from selectors:
$M_i^r := \mathcal{S}^r.wait(\langle \texttt{register}, r, *, * \rangle)$;
(R.2) see if there is a unanimous suggestion from a quorum:
$q_i^r := \mathcal{S}^r.uni\text{-}quorum(M_i^r)$;
(R.3) *register* the suggestion:
$c_i :=$ **if** $q_i^r = \perp$ **then** $(r, \perp)$ **else** q_i^r **fi**;
(R.4) send the suggestion to all deciders:
$\mathcal{D}.broadcast(\langle \texttt{decide}, r, i, c_i \rangle)$

Each **decider** k executes:

(D.1) wait for `decide` messages from registrars:
$N_k^r := \mathcal{R}.wait(\langle \texttt{decide}, r, *, * \rangle)$;
(D.2) see if there is a unanimous suggestion from a quorum:
$d_k^r := \mathcal{R}.uni\text{-}quorum(N_k^r)$;
(D.3) if there is, and not $\perp$, decide:
if ($d_k^r = (r, v')$ **and** $v' \neq \perp$) **then** decide v' **fi**;

Fig. 2. An instance of the consensus protocol.

Proof: Case (a): in the set of suggestions contained in M_i^r collected in Step (R.1) $\mathcal{S}^r.uni\text{-}quorum$ must have identified a unanimous suggestion $q_i^r \neq \perp$ from a quorum of selectors Q_i^r. Consider another honest registrar j that completes Step (R.2) of instance r, computing a unanimous suggestion q_j^r for $q_j^r \neq \perp$ from a quorum Q_j^r. By the consistency property on $\mathcal{S}^r$, $Q_i^r \bigcap Q_j^r$ contains an honest selector. An honest selector always sends the same suggestion to all registrars, implying $q_i^r = q_j^r$.

The case for the deciders (b) is similar although it is based on the consistency property on $\mathcal{R}$, since the deciders require a unanimous suggestion from a quorum of registrars in instance r before deciding. ■

Note that Step (S.2) does *not* have the property of Lemma 1 because selectors do not try to obtain a unanimous suggestion from a quorum.

Corollary 2: In the mechanism in Figure 2, if any honest registrar registers a suggestion (r, v) with $v \neq \perp$ in Step (R.3) of instance r, then any honest registrar that registers a suggestion with non-$\perp$ proposal in that step of that instance, registers the same suggestion.

Proof: By Lemma 1 all honest registrars, unless they crash, compute the same suggestion or $\perp$ in (R.2). Those that computed a non-$\perp$ suggestion therefore register the same suggestion in (R.3). ■

Note that they will also end up sending the same non-$\perp$ suggestion in (R.4) of that instance.

Note that the following lemma holds for all instances other than the first one. Our model does not constraint the initial suggestions the registrars hold.

Lemma 3: In the mechanism in Figure 2, if any honest registrar sends a suggestion $(\bar{r}, v)$ with $v \neq \perp$ in Step (R.0) of instance r then any honest registrar that sends a suggestion $(\bar{r}, v')$ with $v' \neq \perp$ in that step of that instance, sends the same proposal, i.e., $v = v'$.

Proof: By Corollary 2 all honest registrars that register a suggestion in instance $\bar{r}$ based on the same non-$\perp$ proposal register the same suggestion in (R.3). Therefore, if their suggestions sent in (R.0) of instance r were registered with a non-$\perp$ suggestion in (R.3) of the same instance $\bar{r}$, they send the same suggestions. ■

Lemma 4: In the mechanism in Figure 2,

(a) if each honest selector that completes Step (S.4) of instance r sends the same suggestion, then any honest registrar that completes Step (R.2) of that instance computes the same suggestion;
(b) if each honest registrar that completes Step (R.4) of instance r sends the same suggestion, then any honest decider that completes Step (D.2) of that instance computes the same suggestion;
(c) if each honest registrar that completes Step (R.0) of instance r sends the same suggestion, then any honest selector that completes Step (S.2) of that instance computes the same proposal.

Proof: Case (a): assume the conditions of the lemma with respect to each honest selector completing Step (S.4) of instance r. Each registrar that completes Step (R.1) collects suggestions from a maximal-wait set of selectors. By definition, each quorum contains at least one honest selector. Therefore, the set M_i^r of instance r of any honest registrar i cannot contain a unanimous suggestion for a suggestion different from what all the honest selectors have broadcast in Step (S.4). Opaqueness requires that every maximal-wait set contains a quorum of honest participants, therefore, every honest registrar that completes Step (R.2) of instance r will end up having a unanimous suggestion from a quorum of selectors, and will compute the same suggestion.

Case (b) is similar to the first case, and follows because each honest decider will eventually receive `decide` messages for the same suggestion sent by a quorum of honest registrars in Step (R.4).

We now proof case (c) in which each honest registrar that completes Step (R.0) sends the same registered suggestion. Notice that each set of suggestions L_i^r contains at least the suggestions from a guarded set of honest registrars, and cannot contain suggestions from a guarded set of Byzantine registrars. Since each honest registrar sends the same suggestion, no honest selector will find a unanimous suggestion from a guarded set of registrars for any proposal contained in the suggestion of the honest registrars. Each selector will have a unanimous suggestion sent by a guarded set of honest registrars, and therefore will end up computing the same proposal at the end of Step (S.2). ■

And now we address the last and most important property we need to prove:

Lemma 5: The mechanism in Figure 2 satisfies that if r' is the earliest instance in which a proposal w is decided by some honest decider, then for any instance r, $r > r'$, if an honest registrar registers a suggestion in Step (R.3), it is (r, w).

Proof:

Since all instances are taken from a fully ordered set, any subset of them are fully ordered. The proof will

be by induction on all the instances, past instance r', in which eventually some honest registrar registers a suggestion.

Let $w \neq \perp$ be the proposal decided by an honest decider in Step (D.2) of instance r'. Let $Q^{r'} \in \mathcal{R}$ be the quorum in instance r' whose suggestions caused the decider to decide w.

Let $r_1 > r'$ be the first instance past r' at which some honest registrar eventually completes Step (R.3). Since this registrar completes Step (R.3), it must have received `register` messages from a maximal-wait set of selectors following Step (R.1) of instance r_1. Each honest selector that sent such a message received `select` messages from a maximal-wait set of registrars that were sent in their Step (R.0) of instance r_1. Each honest registrar that completes Step (R.0) did not register any new suggestion in any instance r'', $r' < r'' < r_1$, because r_1 is the first such instance. Moreover, the registrar will not register such a suggestion in the future, since it aborted all such instances r'' before sending its `select` message in Step (R.0) of instance r_1.

In Step (R.0) a registrar sends the last suggestion it had registered. Some registrars may send suggestions they had registered prior to instance r' while some other registrars send suggestions they registered in Step (R.4) of instance r'.

Each honest selector j awaits a set of messages L_j from a maximal-wait set in Step (S.1). L_j has to contain suggestions from a quorum Q^{r_1} consisting entirely of honest registrars (by the opaqueness property of $\mathcal{R}$). By the consistency property of $\mathcal{R}$ the intersection of Q^{r_1} and $Q^{r'}$ has to contain a guarded set, and thus Q^{r_1} has to contain suggestions from a guarded set of honest registrars that registered (r', w). There cannot be such a set of suggestions for a later instance, prior to r_1. By Corollary 2 and Lemma 3, there cannot be any suggestions from a guarded set for a different proposal in instance r'. Thus, each honest selector will select a non-$\perp$ proposal and those proposals are identical.

By Lemma 4, every honest registrar that completes Step (R.4) will register the same suggestion, thus the proof holds for r_1.

Now assume that the claim holds for all instances r'', $r' < r'' < r$, and we will prove it for instance r.

There is an honest registrar that completes Step (R.3) in instance r and registers (r, w). Following Step (R.1) of instance r it must have received `register` messages from a maximal-wait set of selectors. Each honest selector that sent such a message received `select` messages from a maximal-wait set of registrars that were sent in Step (R.0) of instance r.

Each honest registrar sends the last suggestion it had registered. Some honest registrars may send suggestions they had registered prior to instance r' while some other honest registrars send suggestions they registered in Step (R.4) of instance r'', $r' \leq r'' < r$. By the induction hypothesis, all honest registrars that send a suggestion that was registered past instance r' use the proposal w in their suggestion.

In instance r, each honest selector j awaits a set of messages L_j from a maximal-wait set in Step (S.1). L_j has to contain suggestions from a quorum Q^r consisting entirely of honest registrars (by the opaqueness property of $\mathcal{R}$). By the consistency property of $\mathcal{R}$ the intersection of Q^r and $Q^{r'}$ has to contain a guarded set, and thus Q^r has to contain suggestions from a guarded set of honest registrars that registered (r', w) in instance r', and may have registered (r'', w) in some later instance. Therefore, selector j obtains w as a possible potential-proposal. Since all honest registrars that register a suggestion past instance r' register the same proposal, there is a support-set for w with associated-instance-identifier $\bar{r} \geq r'$.

There cannot be any other possible potential-proposal with an associated-instance-identifier later than r', since, by induction, no honest registrar registers a suggestion with a different proposal later than r'. Therefore, each honest selector will select the proposal w. By Lemma 4, every honest registrar that will complete Step (R.4) will register the same suggestion, thus the proof holds for r. ■

We now formulate and prove the main theorem about instances:

Theorem 6 (Agreement): If two honest deciders decide, they decide the same proposal.

Proof: If the deciders decide in the same instance, the result follows from Lemma 1. Say one decider decides v' in instance r', and another decider decides v in instance r, with $r' < r$. By Lemma 5, all honest registrars that register in instance r register (r, v'). By the consistency property of $\mathcal{R}$, an honest decider can only decide (r, v') in instance r, and thus $v = v'$. ■

V. Full Protocols

The description of instances above does not specify how instances are created, how broadcasts are done in steps (R.0), (S.4), and (R.4), what specific extended quorum systems to use for $\mathcal{R}$ and $\mathcal{S}^r$, how a selector j obtains proposals for P_j, or how j selects a proposal from P_j. We now show how Paxos [3], the algorithm by Chandra and Toueg [4], and the early protocol by Michael Ben-Or [5] make these choices. They use the instance protocol as a subroutine. For the sake of brevity, we do not include the proofs of the validity and non-blocking properties of these protocols. For a sketch of the proof, please refer to Appendix II.

A. Paxos

Paxos [3] was originally designed only for honest systems. In Paxos, any node can create an instance r at any time, and it becomes the *leader* of that instance. The choice of leader is governed by a weak leader election protocol (see [3]) that is outside the scope of this paper. The leader creates a unique instance identifier r from its node identifier and a sequence number per node that is incremented for each new instance created on that node. The leader runs both a proposer and a selector. $\mathcal{S}^r$ is a leader extended quorum system consisting only of the selector at the leader.

The leader starts the instance by broadcasting a `prepare` message containing the instance identifier to all registrars. (This broadcast is not part of the instance protocol of Figure 2.) Upon receipt, a registrar i first checks that $r > r_i$, and, if so, sets r_i to r and proceeds with Step (R.0). Note that since there is only one participant in $\mathcal{S}_r$, the broadcast in (R.0) is actually a point-to-point message back to the leader, now acting as selector. In Step (S.3), if the leader has to pick a proposal from P_j, it selects the proposal by the local proposer, thus there is no need for proposers to send their proposals to all selectors.

Validity follows directly from the absence of Byzantine participants. To see why Paxos is **Non-Blocking**, consider a state in which some correct decider has not yet decided. Now consider the following continuation of the run: one of the correct nodes creates a new instance with an instance identifier higher than used before. Because there are always correct nodes and there is an infinite number of instance identifiers, this is always possible. The node sends a prepare message to all registrars. All honest registrars start in Step (R.0) of the instance on receipt, so the selector at the leader will receive sufficient `select` messages in Step (S.1) to continue. Due to Lemma 4, and the fact that there is only one selector in $\mathcal{S}^r$, all honest registrars register the same suggestion in Step (R.3). The deciders will each receive a unanimous suggestion from a quorum of registrars in Step (D.1) and decide in Step (D.3).

B. Chandra-Toueg

The Chandra-Toueg algorithm is another consensus protocol that is designed for honest systems [4]. The Chandra-Toueg algorithm requires a coordinator in each instance. The role of the coordinator is similar to the leader in Paxos. However, unlike Paxos, Chandra-Toueg does not use a leader election protocol. Instead, the coordinator of each instance is defined by a simple mod of the instance number by the number of nodes in the system, *i.e.*, the role of the coordinator rotates from node to node at the end of each instance. Each node in the system is both a proposer and a registrar. For each instance r, the selector quorum $\mathcal{S}^r$ is the extended quorum consisting only of the coordinator of that instance.

To start the protocol, a proposer sends a proposal message to all nodes. Upon receiving the first proposal, a registrar starts in instance 0 and executes (R.0). The coordinator of each instance starts (S.0) upon receiving a `select` message for that instance. In (S.3), $P_i.pick(r)$ will simply return the first proposal that was received

by the coordinator. Registrars that successfully complete (R.1-4) move to the next instance.

Note that when registrars are waiting for a `register` message from the selector of a particular instance, they are not guaranteed to receive a reply because the coordinator of that instance can fail. Registrars thus have to be prepared to timeout. When this happens, the registrars starts executing (R.0) in the next instance, which would have a different coordinator. When a registrar receives a `register` message with a larger instance number than the one it is currently in, it aborts the current instance and skips forward to the instance of the `register` message.

In the original description of the Chandra-Toueg algorithm, the coordinator for an instance is also the decider for that instance. This means that in order for all nodes to become aware of a decision, the coordinator has to broadcast an announcement. We can modify the Chandra-Toueg algorithm such that all nodes are deciders in all instances without affecting the rest of the protocol. The effect of this change is the elimination of one round of communication while increasing the number of messages that are sent in (R.4) of the instance mechanism. This is similar to the algorithm proposed in [10]. A comparison of the original Chandra-Toueg algorithm and this modified version was proposed in [11].

As in the case of Paxos, **Validity** follows directly from the absence of Byzantine participants. The **Non-blocking** property follows from that fact that a honest, correct selector can always receive sufficient `select` messages in (S.1) to continue. All honest registrars will always receive the same suggestion in (R.3) since there is only one selector in each instance. If the coordinator for an instance fails, registrars for that instance will timeout and move to the next instance.

C. Ben-Or

In this early protocol [5], each node runs a proposer, a selector, a registrar, and a decider. Instances are numbered $1, 2, ...$ Proposals are either 0 or 1 (that is, this is a binary consensus protocol), and each $P_i = \{0, 1\}$. $P_i.pick(r)$ selects the local proposer's proposal for the first instance, or a random one in later instances.

Each of the selectors, registrars, and deciders starts in instance 1 and runs a loop. A selector j runs a loop consisting of steps (S.1) through (S.4), incrementing r_j right after Step (S.4). A registrar i runs a loop consisting of steps (R.0) through (R.4), incrementing r_i after Step (R.4). Note that the broadcasts in steps (R.0) and (R.4) are to the same destination nodes and happen in consecutive steps, so they can be merged into a single broadcast, resulting in just two broadcasts per instance. Finally, a decider k runs a loop consisting of steps (D.1) through (D.3), incrementing r_k after Step (D.3).

$\mathcal{S}^r$ is the same as $\mathcal{R}$ for every instance r; both consist of all nodes and uses a threshold quorum system. Ben-Or works equally well in honest and Byzantine environments as long as opaqueness is satisfied. It can be easily shown that if a decider decides, all other deciders decide either in the same or the next instance. This suggests an easy termination rule.

Validity follows from the rule that selectors select the locally proposed proposal in the first instance: If all selectors select the same proposal v, by Lemma 4 the registrars register v, and, by the opaqueness property of $\mathcal{R}$, the deciders decide v. The **Non-Blocking** property follows from the rule that honest selectors pick their proposals at random in all but the first instance, and so it is always possible that they pick the same proposal, after which decision in Step (D.3) is guaranteed because of the opaqueness property of $\mathcal{R}$.

VI. Implementation and Protocol Comparisons

The description of the Paxos, Chandra-Toueg, and Ben-Or protocols in the previous section show that while these protocols were originally presented as different from one another, they share a common skeleton. Using the instance mechanism presented in Section IV-E, each of these protocols can be instantiated by using protocol-specific ways of i) defining the selector quorums in each instance, ii) starting instances, and iii) implementing $P_i.pick(r)$ in (S.3).

Having observed the similarities between the three protocols, we now investigate the effect of their differences on their performance. To do this, we implemented

the instance mechanism defined earlier, and built each of the three protocols on top of it.

In this section, we present the implementation of these protocols and results from our simulations.

A. Implementation

We built a replicated state machine out of the consensus protocols that provides a simple logging service to remote clients. The service consists of a set of servers that run a consensus protocol. Clients can submit values to any server, which will then attempt to get that value decided in an *epoch*. To decide a value, a server submits that value as a *proposal* associated with the current epoch. When a value is decided in an *epoch*, the client that submitted the value is informed of the epoch number that the value was decided in, and all servers move to the next epoch. Each server maintains an internal queue of values that it has received from clients and attempts to get them decided in a FIFO fashion.

For the implementation of Paxos, there is an important design decision that was not described in the original protocol [3]. Paxos requires a leader election mechanism that is integral to the performance of the protocol. In our implementation, we support two different leader election mechanisms. First, we built a version of Paxos where each node that wants to propose a value simply makes itself the leader. By having each node pick instance numbers for instances where it is the leader from a disjoint set of instance numbers, we ensure that each instance can only have one unique leader. We call this version of Paxos *GreedyPaxos*.

We also built a second variant of Paxos which uses a token passing mechanism to determine the leader. The details of this protocol are outside the scope of this paper. We call this version of Paxos *TokenPaxos*. in a local variable L_i. Initially, L_i is set to $null$. Upon receiving a valid `prepare` message from another node, *i.e.*, a prepare message for the current epoch and with an instance number larger than any that has previously been received, the node sets the sender of the prepare message as the leader. To propose a value, a node sends the proposal to the current leader. If the current leader is not known, it makes itself the leader and starts an instance. Each `select` message is tagged by the sending registrar with a token request if that node has pending requests in its queue and would like to receive the token in the a subsequent epoch. Each node that sees a register message updates a local bitmap to keep track of the list of nodes which are requesting the token. When the current leader commits all its local requests, it sends the token to a random node that is requesting the token. In order to recover from lost tokens from a crashed leader, each node sets the value of L_i to $null$ if it does not receive any messages from the current leader for a certain amount of time.

For the implementation of Chandra-Toueg, we modify the original algorithm such that all nodes are deciders in all instances. As described in Section V-B, this avoids requiring deciders to broadcast a decision when a value is decided. This improves the performance of our particular application where all servers need to learn about decisions.

All our implementations used a simple threshold quorum system for the selector, registrar, and decider quorums.

B. Experimental Setup

In all our experiments, the logging service consists of a set of 10 servers. The workload is sent to the servers via a set of 10 clients. Each client sends requests to the servers at a rate that is described by a poisson distribution with a mean of λ_c requests per minute. Client's choice of server to send a request to is decided randomly. All messages between client to server and server to server have a latency that is given by a lognormal distribution with a mean of 100 ms and a standard deviation of 20 ms. For each set of experiments, we measure the duration between the time that a server first receives a value from a client to the time that the server learns that the value has been decided.

C. Results

In the first set of experiments, we ran the server against clients with varying loads until 100 values have been decided by the logging service. We vary the request rate

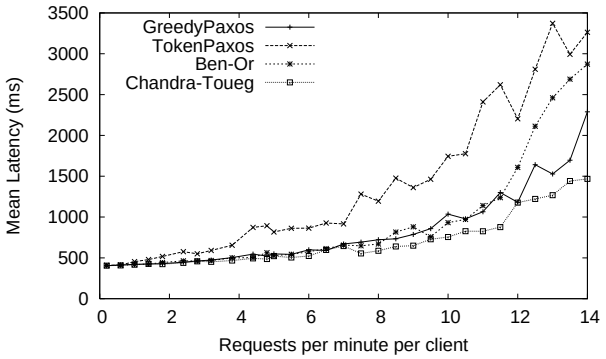

Fig. 3. Mean time to decide a single value under varying request rates

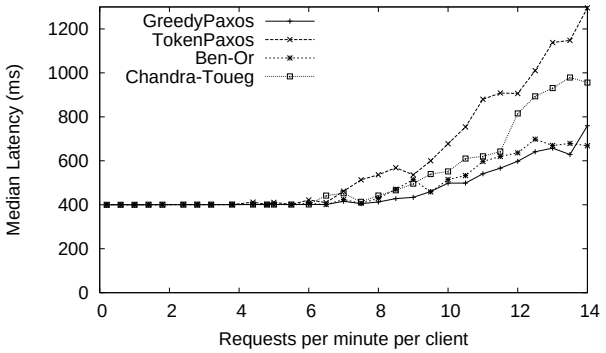

Fig. 4. Median time to decide a single value under varying request rates

from each client, λ_c, from 0.5 requests per minute to 14 requests per minute. We report on the mean and median values of 100 decisions averaged over 8 runs of each experiment. (Presenting full distributions or even just error bars makes the data difficult to read.)

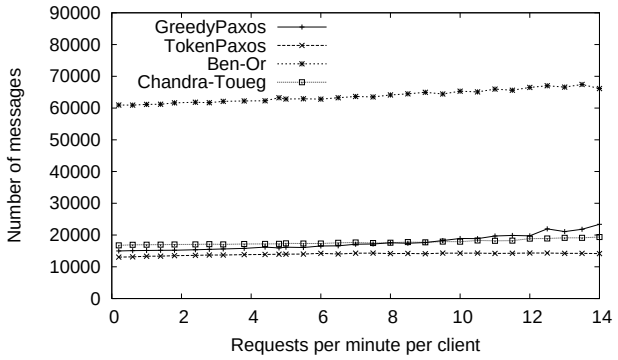

Fig. 5. Communication overhead under varying request rates

Figure 3 and Figure 4 show the mean and the median time it takes for a single value to be decided. The graphs show that as load increases, the time it takes for a value to be decided increases gradually. At low loads, the performance of all four algorithms is about equivalent. This is because in all four algorithms, in the ideal case, it takes four rounds of communication for a value to be decided. This means that in the best case, it takes on average 400 ms for value to be decided.

As load in increased, performance degrades because there is contention between different servers to get different values committed in the same epochs.

Note that GreedyPaxos performs consistently better than TokenPaxos, particularly under heavy load. This is because GreedyPaxos does not need to wait for the token before proposing a value. Under heavy load, each GreedyPaxos node sends a `prepare` message in the beginning of each epoch without having to wait. The node with the largest instance number wins and gets its value decided. TokenPaxos, on the other hand, will always decide values of the node with the token before passing the token to the next node with requests. This has 2 implications: i) if the leader keeps getting new requests, other nodes can starve, and ii) one round of communication is wasted in passing the token. This results in worse performance.

Figure 5 shows the number of messages that each protocol uses to commit 100 values under different request rates. We note that Ben-Or incurs a much larger overhead than the other protocols. This is because Ben-Or uses a selector quorum that consists of all nodes rather than just a leader/coordinator. This means that (R.0) and (S.4) of the instance mechanisms send n^2 messages in each instance, rather than just n messages in Paxos and Chandra-Toueg.

We also observe that compared to TokenPaxos, GreedyPaxos sends more messages as load increases. Under heavy load, each GreedyPaxos node will broadcast a `prepare` message to all other nodes in the beginning of every round. This results in n^2 messages being sent rather than the n `prepare` messages that are sent in the case of TokenPaxos. This effect is shown in Figure 5.

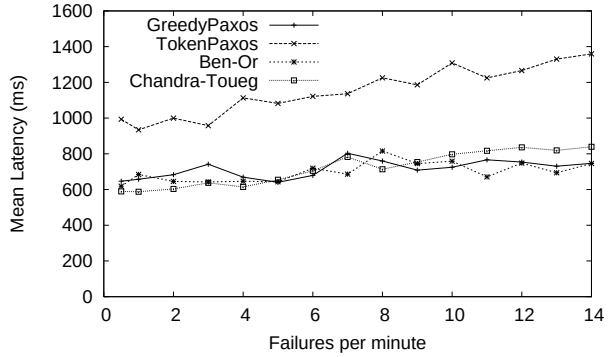

Fig. 6. Mean time to decide a single value under varying failure rates

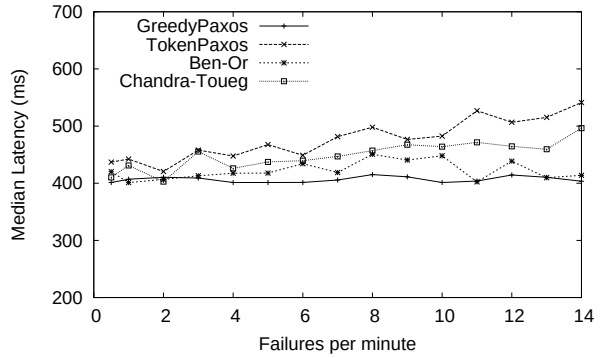

Fig. 7. Median time to decide a single value under varying failure rates

In order to investigate the performance of each protocol under crash failure, we injected failures into our simulations. We modeled the arrival of failure events as a poisson distribution with a rate of λ_f failures per minute. When a failure event occurs, we fail a random server until the end of the epoch. To ensure that the system is able to make progress, we limit the number of failures in an epoch to be less than half the number of servers in the system. Keeping the request rate from clients steady at 7 requests per minute per client, we vary the failure rate from 0.5 failures per minute to 12 failures per minute.

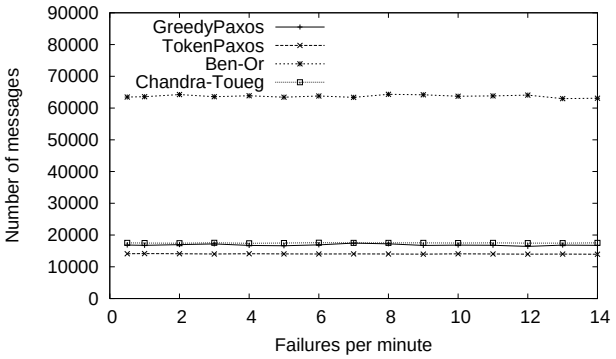

Fig. 8. Communication overhead under varying failure rates

Figure 6 and Figure 7 show mean and the median decision latency, resp., for the the four protocols under varying failure rates. Note that GreedyPaxos and Ben-Or are not affected significantly by server failures. Chandra-Toueg and TokenPaxos, on the other hand, see significant performance degradation as the failure rate increases. This is because Chandra-Toueg and TokenPaxos are both dependent on timeout to recover from failures of particular nodes. In the case of Chandra-Toueg, the failure of the coordinator requires that all registrars timeout and move to the next instance. In the case of TokenPaxos, if the node that is holding the token crashes, a timeout is required to generate a new token.

A comparison study presented by Hayabashibara et al. found that Paxos outperforms Chandra-Toueg [12] under crash failures. We find that this result depends on the leader election protocol used by Paxos. In our experiments, GreedyPaxos outperforms Chandra-Toueg, but TokenPaxos performs worse under certain failure scenarios.

Figure 8 shows the message overhead of each protocol under varying failure rate, clearly showing that the number of messages sent is not affected by failures.

VII. Conclusion

In this paper, we demonstrated that while many well-known consensus protocols such as Paxos, Chandra-Toueg, and Ben-Or at first appear different, they share an underlying commonality. We distilled this commonality into the form of a single skeleton algorithm that can be instantiated into each of these specific protocols by configuring the quorum systems that are used, the way instances are started, and other protocol-specific details. Using this approach, we implemented the skeleton algorithm and used it to instantiate Ben-Or, Chandra-Toueg, and two variants of the Paxos algorithm. Simulation experiments using our implementation allowed us to

explore the effect of the differences between these algorithms on their performance under different workloads and crash failures. We believe that the skeleton algorithm is an interesting basis for the understanding of consensus algorithms and comparison of their performance, and provides a novel platform for the exploration of other possible consensus protocols.

REFERENCES

[1] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.

[2] M. Barborak and M. Malek, "The consensus problem in fault-tolerant computing," *ACM Computing Surveys*, vol. 25, no. 2, 1993.

[3] L. Lamport, "The part-time parliament," *Trans. on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[4] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[5] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols," in *Proc. of the 2nd ACM Symp. on Principles of Distributed Computing*. Montreal, Quebec: ACM SIGOPS-SIGACT, Aug. 1983, pp. 27–30.

[6] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *Trans. on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, July 1982.

[7] M. Fischer, N. Lynch, and M. Patterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[8] D. Malkhi and M. Reiter, "Byzantine Quorum Systems," *Distributed Computing*, vol. 11, pp. 203–213, June 1998.

[9] M. Naor and A. Wool, "The load, capacity, and availabiliity of quorum systems," *SIAM Journal on Computing*, vol. 27, no. 2, pp. 423–447, Apr. 1998.

[10] A. Mostéfaoui and M. Raynal, "Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach," in *Proc. of the International Symposium on Distributed Computing*, 1999, pp. 49–63.

[11] P. Urbán and A. Schiper, "Comparing distributed consensus algorithms," in *Proc. of International Conference on Applied Simulation and Modelling*, 2004, pp. 474–480.

[12] N. Hayashibara, P. Urbán, A. Schiper, and T. Katayama, "Performance comparison between the Paxos and Chandra-Toueg consensus algorithms," in *Proc. of International Arab Conference on Information Technology*, Doha, Qatar, Dec. 2002, pp. 526–533.